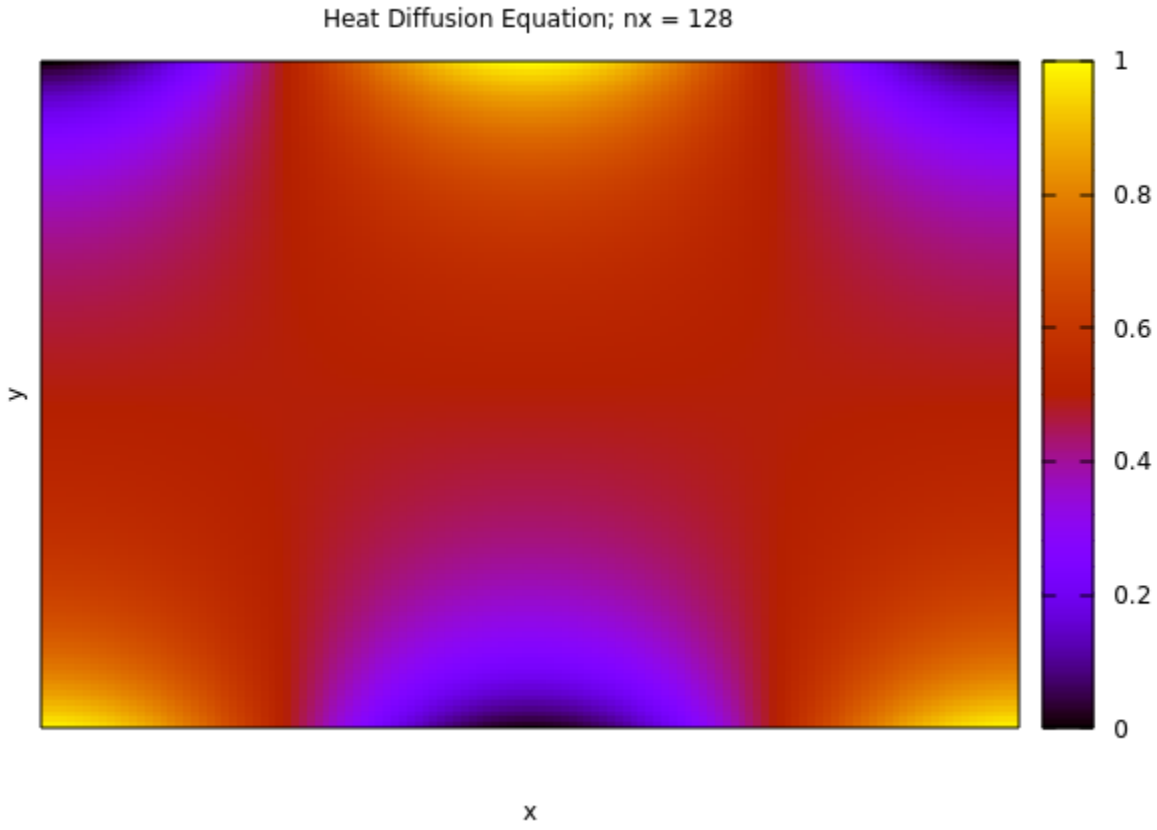
Julienne LaChance
APC 524, HW4

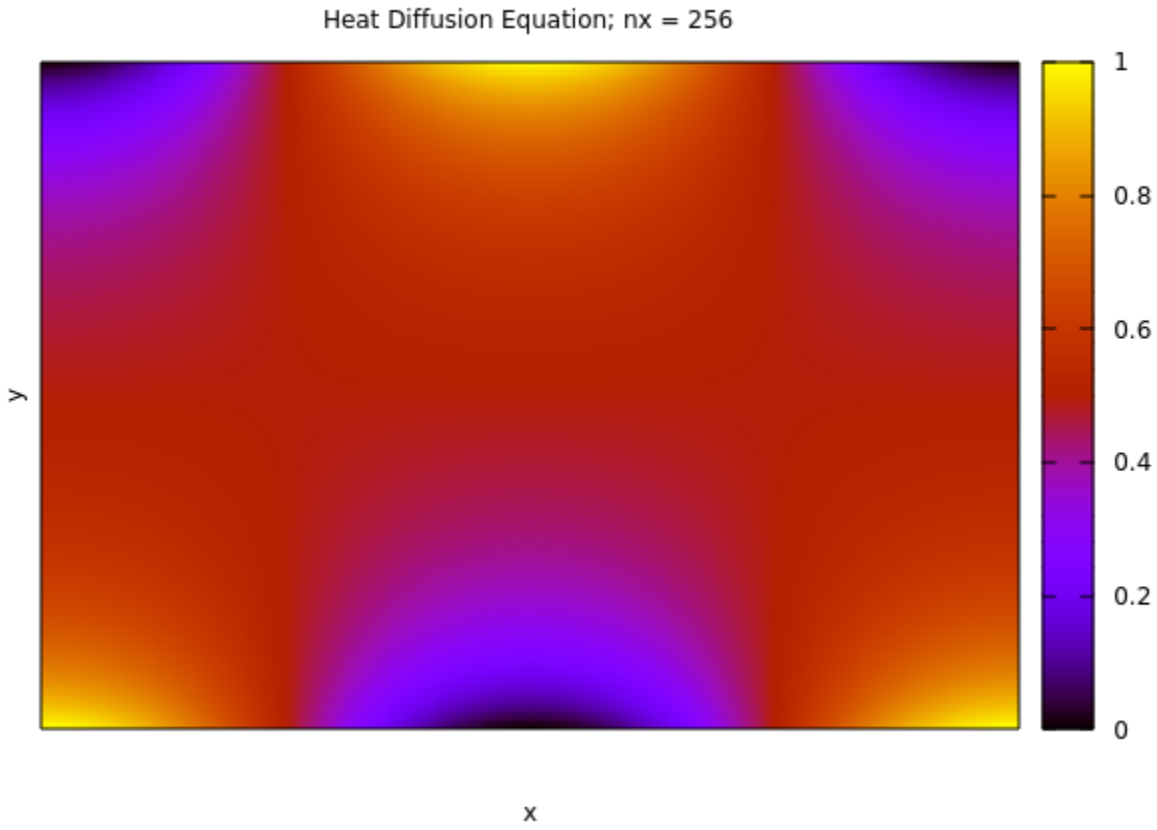
HW4 Summary:

1. Plots and data:
Serial run, contours of final temperature:
For grid size 128^2:



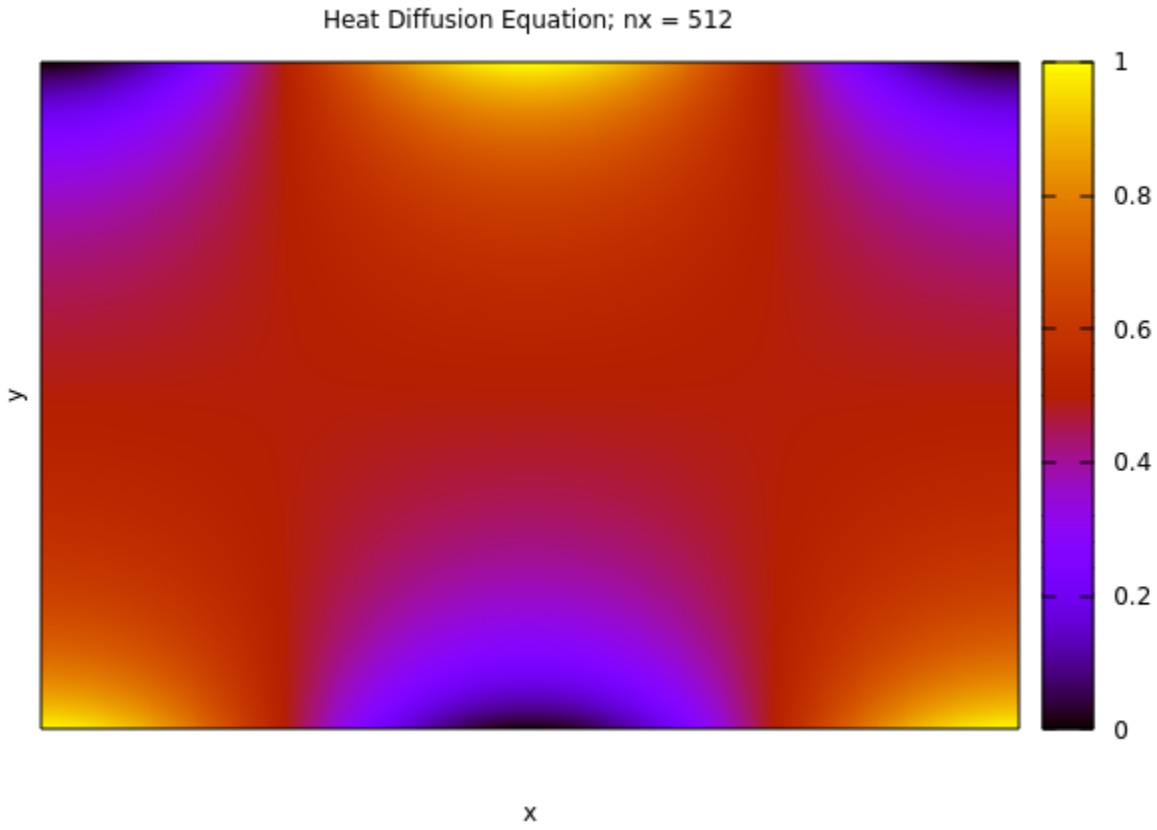
Runtime: 10 s
Volume Averaged Temperature: 0.497063 (boundary conditions included)

For grid size 256^2:



Runtime: 158 s (2.63 minutes)
Volume Averaged Temperature: 0.497074

For grid size 512^2:



Runtime: 2603 s (43+ minutes)
Volume Averaged Temperature: 0.497080

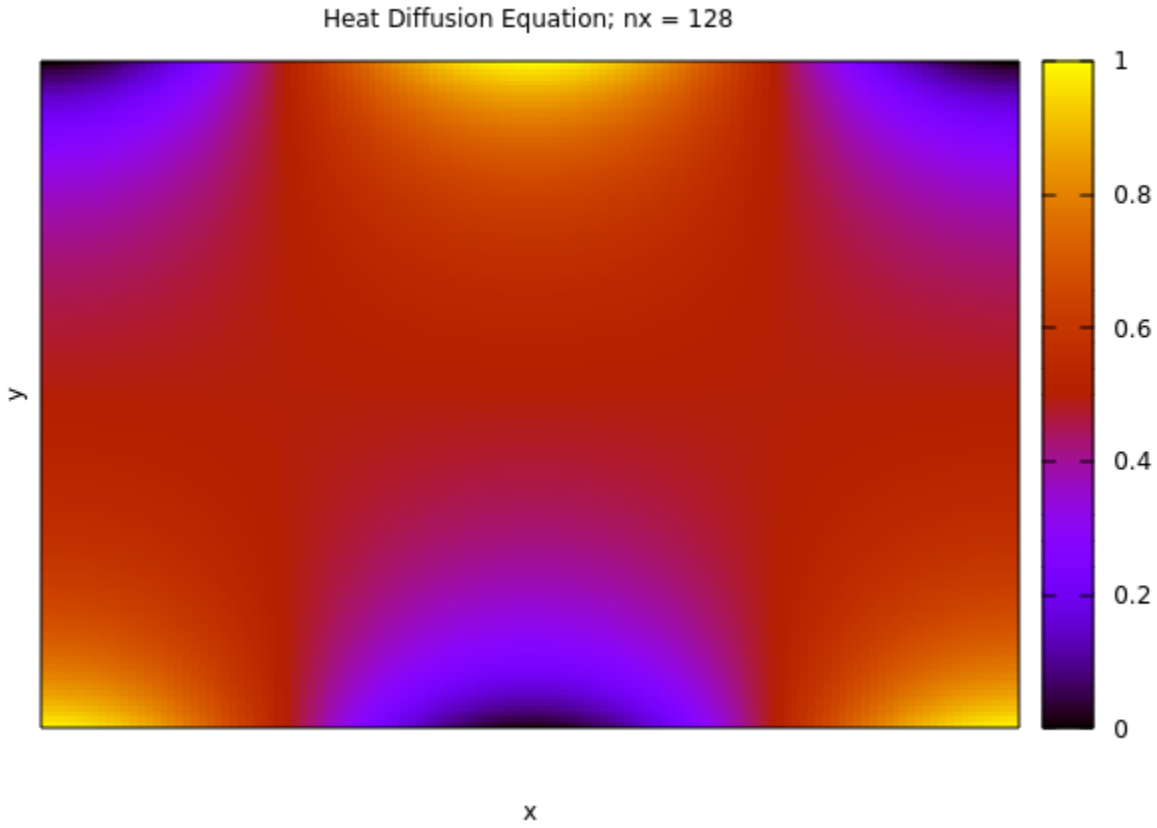
2. Runtimes for various nthread/grid size conditions, in seconds:

	Number of threads in OpenMP run:			
Grid Size:	1	2	4	8
128	16	8	6	3
256	250	129	101	34
512	3924	1993	1625	519

3. Runtimes for various nthread/grid size conditions, in seconds:

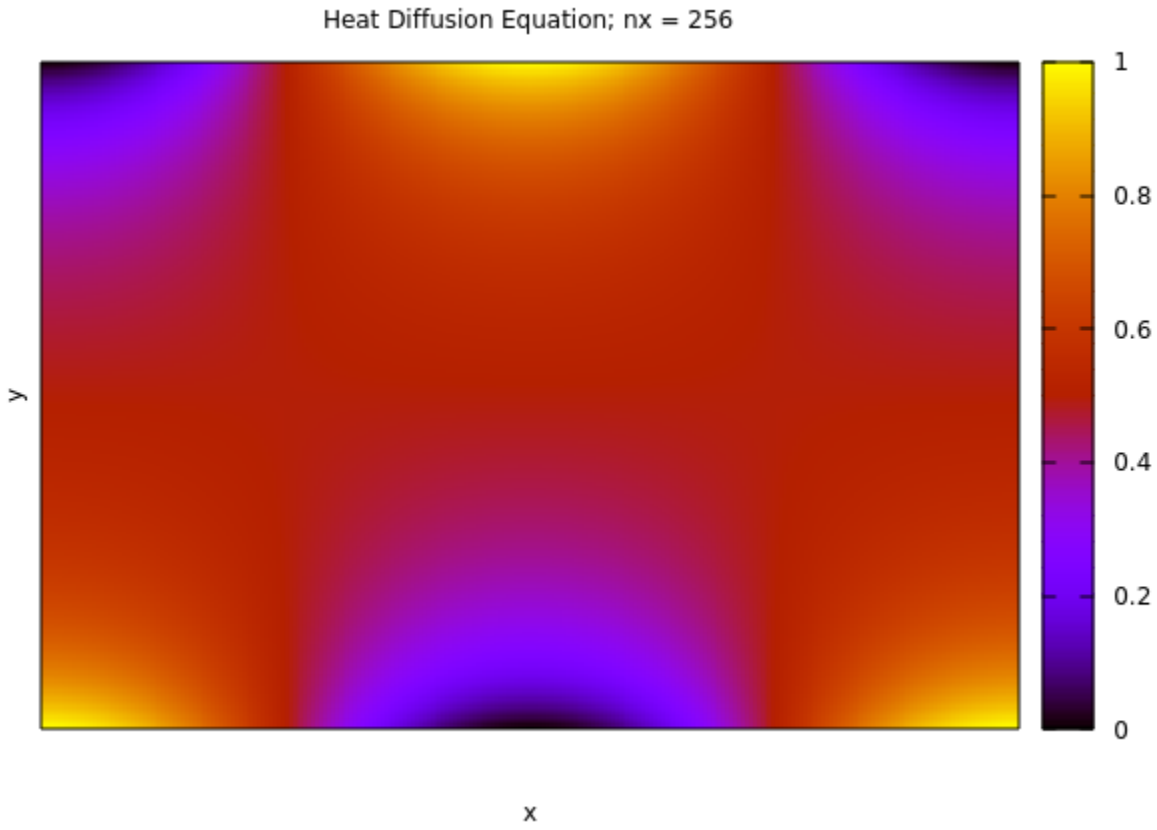
	Number of processors in MPI run:				
Grid Size:	1	2	4	8	16
128	12	3	2	2	2
256	183	23	12	8	5
512	686	227	177	118	73

Plots and data: MPI run, contours of final temperature:
For grid size 128^2:



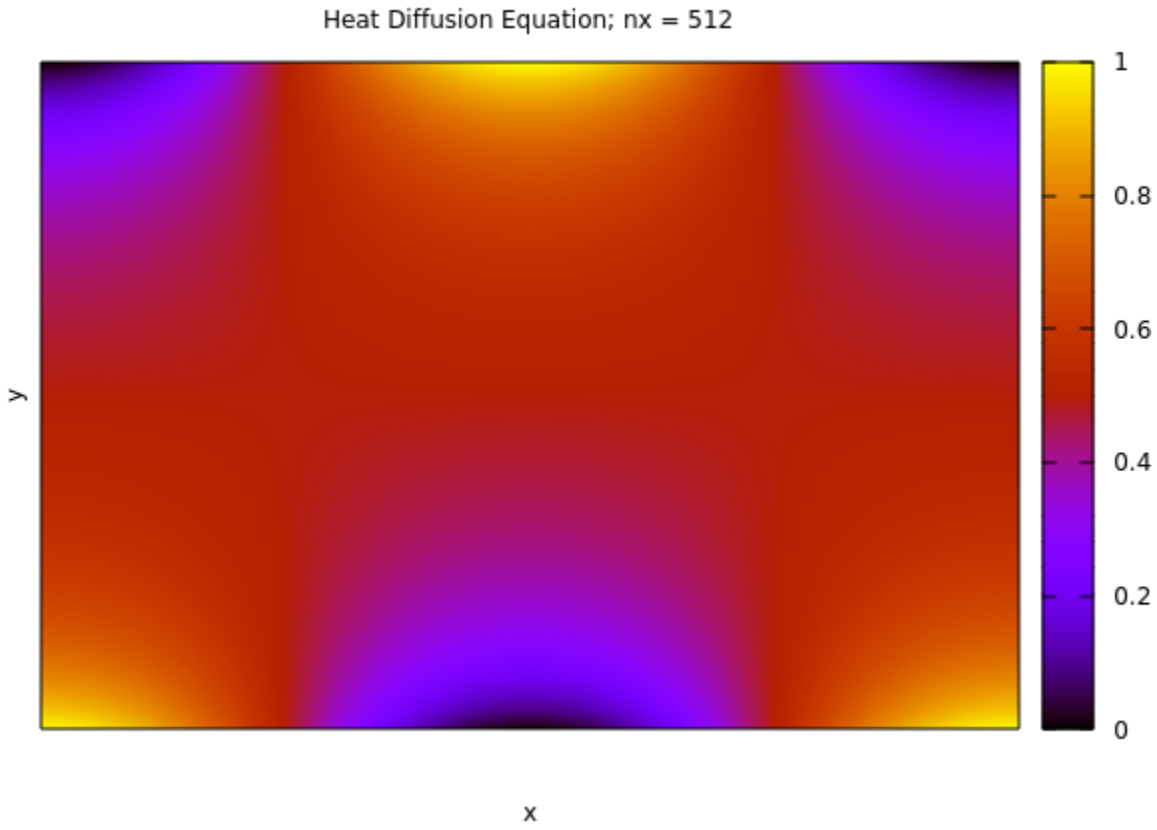
Volume Averaged Temperature: 0.49

For grid size 256^2:



Volume Averaged Temperature: 0.49

For grid size 512^2:



Volume Averaged Temperature: 0.49

Discussion: When implementing this problem in MPI, it was critical to take advantage of domain decomposition- that is, ensuring that we get good load balancing by breaking up our domain into manageable pieces which are evenly distributed over the processors. In order to do this, we had to carefully manage I/O from more than one processor. After initializing the MPI routine, we created sections of the domain with which the current processor was to perform computations. Because the computation of each node in the grid required information from the surrounding nodes, it was necessary to pass columns of data between the processors. This was accomplished using MPI_Send and MPI_Recv calls on “ghost” cell regions to ensure that boundaries were accounted for. Thus each processor was set up to send out and receive several columns of data before proceeding with the temperature grid computations. We also took advantage of the MPI_Allreduce method to sum over grid values over all of the processors and ultimately compute the volume averaged temperature. This method automatically combines values from all processes and distributes values back to all processes- the value in this case being the global sum. Temperature data was ultimately written to individual data files which were combined and plotted using an external script.

4. Discussion:

The major benefit of parallelizing this problem using OpenMP was ease of use. In terms of setup, OpenMP was already available as part of the standard gcc compiler, whereas when using MPI it was necessary to install additional libraries. Additionally, it is much more straightforward to modify the code using OpenMP. The code base was mostly the same but with a few additional compiler directives and calls to the #pragma methods. There was no need to manage communication between threads. However, if it's necessary to set up a very large system, MPI has the advantage in terms of performance. MPI allows us to perform domain decomposition and other methods for evenly distributing work across many processors in an intelligent way- that is, to get good load balancing. MPI allows the user to ensure that threads have roughly the same amount of work and to avoid much of the overhead that comes along with OpenMP. Additionally MPI is portable to any system on which the MPI library has been installed.